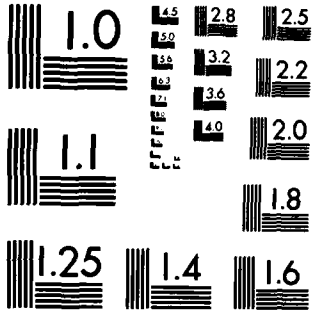
AD-A143 533 A THEORY OF ERROR-BASED TESTING(U) MARYLAND UNIV 1/1
COLLEGE PARK DEPT OF COMPUTER SCIENCE L J MORELL
APR 84 TR-1395 AFOSR-TR-84-0563 F49620-83-K-0010
UNCLASSIFIED F/G 9/2 NL

END
FILMED
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(4)

Technical Report TR-1395 *April, 1984*

A Theory of Error-based Testing

Larry Joe Morell

Department of Computer Science
University of Maryland
College Park, MD 20740

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS:
2a. SECURITY CLASSIFICATION AUTHORITY:	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE:	
4. PERFORMING ORGANIZATION REPORT NUMBER(S):	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 84-0563
6a. NAME OF PERFORMING ORGANIZATION: University of Maryland; 6b. OFFICE SYMBOL (If applicable):	7a. NAME OF MONITORING ORGANIZATION: Air Force Office of Scientific Research
6c. ADDRESS (City, State and ZIP Code): Department of Computer Science, College Park MD 20742	7b. ADDRESS (City, State and ZIP Code): Directorate of Mathematical & Information Sciences, Bolling AFB DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR; 8b. OFFICE SYMBOL (If applicable): NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F49620-83-K-0018
8c. ADDRESS (City, State and ZIP Code): Bolling AFB DC 20332	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
61102F	2304	A7	

11. TITLE (Include Security Classification): A THEORY OF ERROR-BASED TESTING.

12. PERSONAL AUTHOR(S): Larry Joe Morell*

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	APR 84	81

16. SUPPLEMENTARY NOTATION: *After 1 Sep 84, address will be Dept of Mathematics & Computer Science, College of William and Mary, Williamsburg VA 23185.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Testing is the process of inferring the correctness of a program based on information collected during program execution. This process is called error-based when the information collected is used to infer that certain errors are not in a program. It is assumed here that a program can only be incorrect in a limited fashion specified by associating alternate expressions with program expressions. Substitution of an alternate expression for a program expression yields an alternate program that is potentially correct. The goal of error-based testing is to differentiate the program from each of its alternates.

Two types of error-based testing are given. In static error-based testing the code is analyzed to produce two conditions. A creation condition describes when a class of alternate expressions will affect the program state. A propagation condition describes when an alternate program state will affect the output. If the output of the program is correct and both conditions are satisfied, all alternatives are faults detected by the test set. (CONT.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Dr. Robert N. Buchal	(202) 767- 4939	NM

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION

ITEM #19, ABSTRACT, CONTINUED: In dynamic error-based testing information is first collected from text executions. It is then proved that certain alternate programs are eliminated. A particular form of dynamic error-based testing based on symbolic execution is presented. In symbolic testing program expressions are replaced by symbolic alternatives that represent classes of alternate expressions. The output from the system is an expression in terms of the input and the symbolic alternative. Equating this with the output from the original program yields a propagation equation whose solutions are those alternatives which are not differentiated by this test.

Since an alternative set can be infinite, it is possible that no finite test differentiates the program from all its alternates. Circumstances are described as to when this is decidable.

Program faults can interact in such a way that error-based testing declares each correct in the presence of the others. Such interaction is shown to occur rarely for a limited class of programs.

Technical Report TR-1395 April, 1984

A Theory of Error-based Testing

Larry Joe Morell

Department of Computer Science
University of Maryland
College Park, MD 20740

This work was partially supported by the Air Force Office of Scientific Research, under contract F49620-83-K-0018. Author's address after September 1, 1984: Department of Mathematics and Computer Science, College of William and Mary, Williamsburg, VA 23185.

ABSTRACT

Testing is the process of inferring the correctness of a program based on information collected during program execution. This process is called *error-based* when the information collected is used to infer that certain errors are not in a program. It is assumed here that a program can only be incorrect in a limited fashion specified by associating *alternate expressions* with program expressions. Substitution of an alternate expression for a program expression yields an *alternate program* that is potentially correct. The goal of error-based testing is to differentiate the program from each of its alternates.

Two types of error-based testing are given. In *static* error-based testing the code is analyzed to produce two conditions. A *creation condition* describes when a class of alternate expressions will affect the program state. A *propagation condition* describes when an alternate program state will affect the output. If the output of the program is correct and both conditions are satisfied, all alternatives are *faults* detected by the test set.

In *dynamic* error-based testing information is first collected from test executions. It is then proved that certain alternate programs are eliminated. A particular form of dynamic error-based testing based on symbolic execution is presented. In *symbolic testing* program expressions are replaced by symbolic alternatives that represent classes of alternate expressions. The output from the system is an expression in terms of the input and the symbolic alternative. Equating this with the output from the original program yields a *propagation equation* whose solutions are those alternatives which are not differentiated by this test.

Since an alternative set can be infinite, it is possible that no finite test differentiates the program from all its alternates. Circumstances are described as to when this is decidable.

Program faults can interact in such a way that error-based testing declares each correct in the presence of the others. Such interaction is shown to occur rarely for a limited class of programs.

TABLE OF CONTENTS

TABLE OF CONTENTS

1. Introduction

Testing is the process of inferring the correctness of a program from information deduced from program executions. The role of testing theory is to study this process and to determine the conditions under which correctness can be shown. Testing may be viewed in two stages. First, a *testing strategy* is developed which describes how testing is to be performed; i.e. how the input data is to be selected, what information is to be collected, and how to collect and analyze this information. Second, there is the application of the strategy to a given program, resulting in a *test* of that program.

A testing theory must provide a means of judging the quality of either stage of testing. Intuitively, a test is *good* if it provides convincing evidence that a program is correct. A testing strategy could therefore be considered good if it produces good tests when applied to a particular program. This approach has been taken by Howden [How76] who defines a *reliable* test to be one whose success implies program correctness. The disappointing result, however, is that a reliable test is not attainable in general. It is thus necessary to try a different approach to defining the quality of a test or a testing strategy.

The intent of a quality measure is to determine when one test (or testing strategy) is better than another. It is therefore desirable to have gradations of goodness, with a reliable test being the ultimate. Structural coverage measures such as statement coverage, branch coverage, or path coverage [Mil74] do not satisfy such a gradation. Even when these criteria are satisfied they do not imply correctness. For example, executing all statements does not guarantee that a

program is correct.

The means taken here of assessing the quality of tests is to compare them on the basis of how many errors they demonstrate are not in the program, i.e., on how many errors each test *eliminates*. This choice satisfies the gradation mentioned above since the highest quality test by this approach is one which eliminates all errors, implying the program is correct. For testing strategies a conservative approximation is taken to be that of the weakest test that satisfies the strategy. The quality of a strategy is therefore determined solely by what errors it is guaranteed to eliminate.

Testing is *error-based* when it is performed with the aim of eliminating errors. This notion was introduced by Weyuker and Ostrand [Wey80]. In their approach, a program's input is partitioned into a set of disjoint classes called *subdomains*. These subdomains are then shown to be *revealing* for certain errors; i.e., if any member of a subdomain reveal a designated error, then all members of the subdomain reveal the error. In this manner an error is eliminated if the program is correct for an input from a subdomain that is revealing for that error. Only specified errors are eliminated.

The difficulty with Weyuker's and Ostrand's approach is that it does not specify the means of designating errors and hence it is impossible to systematically apply their ideas or to prove any properties of what they envision as error-based testing. The means of eliminating errors is therefore explained by example rather than by providing a method to follow independent of the error categories. They do indicate, however, that for every supposed error in the

program, test data must be developed that eliminates the error. In order to do this a proof must be given that certain inputs eliminate particular errors. What is unclear is whether there is any general technique for doing such proofs or what exactly constitutes a proof. There is also no comment on whether such proofs are always possible, and if not, when they are possible. It is therefore impossible to determine for what categories of errors the proofs can be automated.

This dissertation attempts to develop an error-based testing meta-theory that

(1) defines error-based testing,

(2) provides strategies by which error-based testing can be applied, and

(3) proves properties about the strategies concerning what errors can and cannot be eliminated.

The theory defines the mechanism whereby the errors can be defined, and once defined, how they are eliminated. It is a meta-theory in the sense that it applies once the error categories have been chosen, and is therefore independent of any particular class of errors.

The model chosen here for what is an error is adopted from mutation testing [DeM78]. Expressions are supposed incorrect in a limited manner as defined by a set of *alternatives*. This set contains legal replacements for the expression. In mutation testing inputs are found for which the original program is correct and for which all alternate programs (those induced by substituting an alternative for a program expression) are incorrect. This is done by generating each alternate

program and then interpreting each one for the test data. The resulting output is then compared to the output of the original program for the same data. If they differ and the original program is correct, then the alternate program is eliminated; otherwise, either the alternate program is equivalent to the original program or the test data must be augmented to eliminate the alternate program.

The mutation model of errors is extended here in two ways. First, alternative sets are allowed to be infinite. This entails developing an error-based testing theory that does not rely on the ability to generate and execute all alternatives. The second extension involves the analysis of compound errors. Mutation testing assumes that simple errors are coupled to complex errors in such a way that the elimination of simple errors implies the elimination of complex errors [DeM78]. The terms simple and complex are undefined, however, so this assumption cannot be verified. A precise albeit restricted version of this assumption is given here. A complex error is considered to be a combination of simple errors. Error-based testing then is the process of proving that every alternative from every alternative set is eliminated by a test set and that all combinations are likewise eliminated.

Error-based testing therefore intertwines elements from both program verification and traditional testing. This approach varies considerably from how testing and verification have been related in the past. The tendency has been to place testing and verification as separate activities, as antagonists in the game of demonstrating program correctness. For example, [Ger76] suggests that testing is useful after a formal proof to check the veracity of the proof. [Goo75]

proposes that testing is useful in establishing the basis step in an inductive proof. [Gel78] suggests that testing can be used to prove assertions which would otherwise be tedious to prove by hand. There has been no concerted attempt to combine testing and verification into a unified theory.

The error-based testing theory developed here integrates testing with program verification by exploiting the advantages of both. Properties are proved about the program or its alternative sets which in conjunction with program execution imply that certain errors are not present in the program. This process is illustrated below.

The simplest way to eliminate an alternate program is to find an input on which the original and the alternate differ. If the original is correct for that input, then the alternate must be incorrect, and is therefore eliminated. What is not obvious from this argument is that the alternate program need not be executed (as in mutation testing); it need only be shown that if it were executed, a different output would result. For example, it may be clear from the program text that an alternate program would follow a path which guarantees a different output. Consider the program

```
if a**2 + b**2 = c**2
   then print "right triangle"
   else print "not right triangle"
```

in which three alternate programs can be formed by replacing the equality operator with any inequality operator ($<$, $>$, $\neq$). One provable property of this program is that if a different branch is taken then a different output results. But

in each alternate program a branch different than the original will be taken for all values of a, b, and c for which the condition is true. There is no need to execute any alternate to eliminate it; a property of the code combined with the appropriate execution of the original is sufficient to guarantee that each is eliminated. The process above can be seen in incipient form in [How82] where properties of a program statement are used to deduce when a change to the statement would locally affect the program's computation. This is called *weak mutation testing* because it is possible that a local effect will not be manifested in a program failure, thus the alternate program would not be eliminated. Weak mutation testing is extended here by proving when a local effect will be manifested in the program output.

As another example, consider computing the average of three numbers:

ave := (i + j + k) / 3

Suppose the alternate programs are those in which the divisor 3 is replaced with some other constant. For the inputs i = 6, j = 2, and k = 7 the program computes 5 if and only if the divisor is 3. If the constant 3 is replaced by any other constant in this statement, a different answer results for this input. Thus, infinitely many alternate programs are eliminated without executing any of them.

Corresponding to these examples are two ways to eliminate alternate programs without additional executions. The first is code-based and is called *static* error-based testing. Properties of the code are proved which guarantee that the introduction of specified alternatives would produce different outputs for

particular inputs. Tests are then developed which produce those inputs. If the outputs are correct, then the alternate programs are eliminated. The second method of elimination is execution-based and is called *dynamic* error-based testing. Information is collected about the program computation for a test set. Based on this information, it is proved that certain faults could not have been in the program. This then eliminates the corresponding alternative programs.

It may be objected that static and dynamic error-based testing wastes time since the alternate programs could be automatically generated and executed, eliminating each alternate separately. This mutation process has three defects, though. The first is the need for an executable oracle. Though this may be assumed for theoretical purposes (as is done here) it is frequently not available in practice. Thus, when the outputs from the original and the alternate programs differ, how is it decided which is correct? For a few cases, a person could check for correctness, but not for the multitude of cases presented by mutation testing. Furthermore, an automated system cannot decide equivalence and therefore requires human monitoring. A second defect is that such an automated system assumes that there are finitely many alternate programs. As mentioned above, no such assumption is made here. The third difficulty is one of efficiency rather than theory. Generating and executing millions of alternate programs may consume more resources than desired. Furthermore, the expended effort is of no use in testing other programs.

Section 2 gives a basic theory of testing and demonstrates the need for error-based testing. Section 3 then defines error-based testing and discusses the

issue of coupled errors.

Sections 4 and 5 discuss static and dynamic error-based testing, respectively. Particular emphasis is given to the decidability of when an infinite alternative set can be eliminated by a finite test. For a simple programming language this is analyzed in depth for alternatives which are substitutions of one constant for another. It is shown for such constant substitutions that the dividing line between decidability and undecidability is whether or not the substitution affects the program flow. If it does not then a finite test exists which eliminates all alternate programs. If the substitutions do affect the program flow then it is possible that no finite test set eliminates all of them; even worse, it is undecidable when this occurs.

Section 6 relates the present work to the literature. The last section summarizes the dissertation and suggests future directions of research.

2. A general theory of testing

A general theory of testing should be both descriptive and prescriptive. It should define the components of testing and should provide convenient terminology and notation for discussing them. It should also define the relationships among the components, showing what is solvable by means of testing and what is not. For those areas in which testing is weak, a general theory should indicate the direction of more specialized work. This section attempts such a theory.

2.1. Basic components and relationships

The fundamental components in testing are programs, specifications, and test sets.

Definition A *relation* is a set of ordered pairs. If R is a relation then its *domain* is $\{x \mid (x,y) \in R\}$ and is written as dom(R).

Every program defines a relation called its *program relation* defined as

$\{(x,y) \mid$ program P on input x halts with output y$\}$.

Since programs and their corresponding relations are discussed frequently, it is useful to adopt a notation that distinguishes the two. [Lin79] suggests the following: If P is a program, then its program relation is denoted by [P]. If $(x,y) \in [P]$ and $(x,z) \in [P] \Rightarrow y = z$ then the relation is a function and the program is called *deterministic*. The theory developed below is for deterministic programs, but can be readily extended for non-deterministic programs. Since [P] is a

function, [P](x) is the output P computes on input x. If no such output exists, then [P] is *undefined* at x and this is written as [P](x)↑; otherwise [P] is defined on x which is written as [P](x)↓.

A *specification* states what a program should compute. Since a program is a set of ordered pairs, it is reasonable for a specification to be a set of ordered pairs. There are, however two additional properties a specification should have.

Definition A set is *recursive* if there is an algorithm for deciding whether or not any element is a member of the set. (The intuitive notion of algorithm is made precise by several formal computation models; see, for example [Bra74].)

These properties are: (1) The domain of a specification must be recursive. This avoids the awkward circumstances of not knowing whether a program is supposed to halt on a given input. (2) The specification must be recursive. This enables the specification to act as an *oracle*, a decision procedure which can decide whether or not any proposed input-output pair is in the specified relation. Neither property implies the other as shown by the following theorems.

Theorem There is no program, P, which computes the following function:

$$[P](x) = \begin{cases} 1 & \text{if } [x](x)\downarrow \\ 0 & \text{if } [x](x)\uparrow \end{cases}$$

where x is a program.

Proof If P exists then it is then possible to construct

$$[P'](x) = \begin{cases} 1 & \text{if } [x](x)=0 \\ 0 & \textit{otherwise} \end{cases}$$

by replacing every print statement in P by a conditional computationally equivalent to the following:

```
if output = 1 then
  if [x](x) = 0
    then output := 1
    else output := 0
else output := 0
print output
```

Since P is defined everywhere, so is P′. In particular P′ should be defined on itself. This, however, yields an immediate contradiction, because [P′](P′) must be either 0 or 1 by definition, but can be neither because [P′](P′) = 0 if and only if [P′](P′) = 1. This Russell-like paradox disproves the assumption that the program P exists.

Corollary There exist non-recursive relations with recursive domains.

Proof The set K = {x | x is a program, [x](x)↓} is not recursive as a direct result of the proof above. The relation {(x,y) | x = 1 & y ∈ K} is a non-recursive relation with a recursive domain.

Theorem There exist recursive relations with non-recursive domains.

Proof Let S = {(x,y) | [x](x) halts in fewer than y steps}. Clearly, S is recursive because an interpreter can be written which will execute program x on itself as input for up to y steps, inspecting to see if the program has halted. But dom(S) = K, which is not recursive.

The central concept relating programs to specifications is that of correctness.

Definition [Mil80] A program P is *correct* with respect to a specification S if and only if dom ([P] $\cap$ S) = dom(S).

Thus, a program is correct with respect to a specification if and only if it computes results for the entire domain of interest, and all the results it computes are as specified.

Since programs and specifications are frequently associated in testing, it is useful to encapsulate the components testing relates.

Definition An *arena* is a 3-tuple <P,S,D> where P is a program, S is a specification, and D = dom(S).

The third component is redundant, but is included for notational convenience so the domain of the specification need not be continually named. The definitions above and below can be easily altered to allow the third component to be a subset of the domain of the specification.

Much attention has been given to demonstrating the correctness of a program with respect to a specification. There are two general approaches to this problem, program verification and testing. In program verification, a formal proof is given to show that the program computes what is specified. Different specification methods induce different verification techniques, but all are based on information derived from the program code rather than from execution

information [Lin79] [Hoa69]. In testing, correctness is deduced from information derived from program executions. Many formally "proved" programs have been shown later to be incorrect, often by executing a few test cases [Ger76]. As shown below, testing can never prove correctness, but it can check restricted hypotheses about a program. The direction of this dissertation is to exploit the strengths of both approaches to verifying correctness.

The final component of the testing theory presented here is a test set.

Definition A *test set*, T, for an arena $G = <P,S,D>$ is a subset of D. T is *halting* if and only if $T \subseteq dom([P])$. T is finite unless otherwise stated.

2.2. An alternate approach to correctness

An alternate but equivalent way of demonstrating correctness is by showing that the program is not incorrect. Thus, it might be hoped that as the test set grows, more information is gained about when the program fails.

Definition For an arena $G = <P,S,D>$, $x \in D$ is *successful* if and only if $[P](x)\downarrow$ and $(x,[P](x)) \in S$, otherwise x is a *failure*. A test set T for G is successful if and only if each $x \in T$ is successful for G. The set of all successful points for G is called the *successful set* of G. The set of all failure points for G is called the *failure set* of G.

Definition The *defined failure set* of an arena G is all members of D for which [P] is defined. The *undefined failure set* of an arena G is all members of the

failure set of G for which [P] is undefined.

Definition A halting test set T for an arena G = <P,S,D> is *reliable* if and only if: T successful for G implies P correct with respect to S.

Exploring the relationship between testing and program correctness may be viewed as investigating the characteristics of the failure set. Can the failure set be a random collection of points, or must it conform to some pattern? The most general pattern would be those sets which are the domain of a program. Such sets are called *recursively enumerable.* If the failure sets are not recursively enumerable there is little hope in discovering anything decidable.

Theorem The failure set of an arena is not necessarily recursively enumerable.

Proof Assume that the failure set of every arena is recursively enumerable. Let I denote the set of integers. Let G_P = <P,S,D> be a collection of arenas in which S = I x I, one for each program P. (Thus, S is all possible ordered pairs, so in these arenas any computed output by the program is correct.) For each arena in this collection the complement of the failure set is the halting set, which is by definition recursively enumerable. The assumption is that each failure set is recursively enumerable. But if a set and its complement are both recursively enumerable, then they are both recursive [Bra74]. So, for any program P, it is decidable whether or not [P](P) is defined, that is, K = $\{x \mid [x](x)\downarrow\}$ is recursive, which it is not. Therefore, the failure set of an arbitrary arena is not recursively enumerable.

When faced with trying to deduce information about non-recursively enumerable sets, not much can be done. For example,

Theorem A reliable test set exists but can neither be generated nor recognized for an arbitrary arena.

Proof If a program is correct then any subset of the domain of the specification is a reliable test set. If the program is incorrect, then there must be some input x in the failure set. The set {x} is therefore vacuously reliable. Hence a reliable test set always exists. Suppose a reliable test set can be generated for an arbitrary arena. Then a program, R, can be constructed for the function which determines program equivalence

$$[R](P,S) = \begin{cases} 1 & \text{if } [P]=[S] \\ 0 & \textit{otherwise} \end{cases}$$

by generating a reliable test set for P with respect to [S], executing P, and checking agreement with [S]. If they agree, then they are equivalent; produce 1. Otherwise, produce 0. However, if program equivalence is solvable, then it would be decidable whether or not an arbitrary function is the zero function. In particular, it would be possible to determine if

$$f(k) = \begin{cases} 1 & \text{if } [P](P)\downarrow \textit{ in less than } k \textit{ steps} \\ 0 & \textit{otherwise} \end{cases}$$

for each P is everywhere zero, which means that the set K would be recursive, which it is not. Thus, a reliable test set cannot be generated. It cannot be recognized either, because a generator could then be constructed by producing successively larger test sets until the recognizer declares one to be reliable.

2.3. Research direction

To progress beyond this state of affairs it is necessary to make some assumptions about the failure sets of the arenas under consideration. For example, some positive results can be obtained if it is possible to decide when the program halts.

Theorem For any arena $G = \langle P,S,D \rangle$, if dom([P]) is recursive then the failure set, defined failure set, and the undefined failure set of G are recursive.

Proof Since dom([P]) is recursive it is decidable whether or not P is undefined for any input x. Thus, the undefined failure set is recursive. If P is defined for an input x, then x is either in the successful set or in the defined failure set. To decide which, compute [P](x). Since the specification S is recursive, it is decidable whether or not $(x,[P](x)) \in S$ for any $x \in dom([P])$. If $(x,[P](x)) \in S$ the x is a member of the successful set, otherwise it is a member of the defined failure set. Since both the defined and undefined failure sets are recursive, so is their union (the failure set)

Making assumptions about the failure set has the effect of limiting the class of programs being considered to meet a given specification. Without any assumptions, testing must distinguish the original program from all programs. This is impossible, as shown by the above result on reliable test sets. However, by suitably restricting the class of programs (and implicitly restricting the resulting failure set), testing can demonstrate correctness under the assumptions.

One place to begin is with the programming language definition. The programming language may be restricted to only a subset of all computable functions. Since many undecidable problems result for unbounded loops, these could be disallowed in the language. Halting problems would no longer plague the tester. Even under such assumptions the class of programs are more complex than can be proved correct by testing [Jon77]. For an even more restrictive language [Tsi70], but one which is of limited practical value, reliable tests can be generated.

The class of programs can be restricted by specifying an upper bound on size. The program under test could be compared to all programs which are smaller than it by some measure. The goal of testing is then to distinguish the original program from all non-equivalent smaller programs.

The class of alternate programs can also be specified by defining transformations to apply to the program. By chosing the transformations to represent typical programmer mistakes, the goal of testing is then to show that that the programmer did not make any of these mistakes. Such testing is called error-based testing and is the method formalized and discussed in the following sections.

3. Error-based Testing

The appeal of error-based testing is that it is closely related to the alternate way of showing program correctness given in section 2: show that the program is not incorrect. If a program has limited potential for being incorrect, then a test set demonstrates correctness when it shows the potential is not realized.

It is first necessary to distinguish among the various forms of incorrectness. The IEEE conventions [IEE83] are adopted here, with slight variation.

Definition An *error* is a mental mistake by a programmer or designer. It may result in textual problem with the code called a *fault*. A *failure* occurs when a program computes an incorrect output for an input in the domain of the specification.

For example, a programmer may make an error in analyzing an algorithm. This may result in writing a code fault, say a reference to an incorrect variable. When the program is executed it may fail for some inputs.

The means of specifying incorrectness taken here is to define potential faults for program constructs. Thus the subject is fault-based testing, but the existing literature calls this error-based testing, and hence that term is used.

3.1. Basic definitions

Definition A *location* in a program isolates some expression of the program. "Expression" is necessarily vague since it is language dependent. An *alternate expression* (*alternative*) is an expression f which can be legally substituted for the

expression designated by l. The resulting program is called an *alternate program* and is denoted by P_f^l. $P_{f,g}^{l_i,l_j}$ denotes the *multiple* alternate program formed by substituting expressions f and g in locations l_i and l_j, respectively. A set of alternatives for an expression is called an *alternative set.* An alternative set always contains the original expression unless otherwise stated.

An alternative set is used to represent potential faults in a program. It may be defined by enumeration, or by giving an algorithm by which the members may be generated. The method of description will usually be an English phrase. For instance, rather than giving an algorithm for generating all prime numbers, the phrase "all prime numbers" would be used. However, if any doubt may arise, the algorithm is given.

Definition An *error-based arena* is a 5-tuple, $\mathbf{E} = <P,S,D,L,A>$, where $<P,S,D>$ is an arena, $L = (l_1, l_2, \ldots, l_n)$ is an n-tuple of locations in P, and $A = (A_1, A_2,\ldots,A_n)$ where each A_i is an alternative set associated with location l_i, $1 \leq i \leq n$. If each alternative set is finite then **E** is called *bounded*, else it is *unbounded.* The set of all alternate programs for **E** is those programs generated by substituting any alternative in its respective location in the program P. It is denoted by $P_{\mathbf{E}}$. The program P is called the *original* program. When discussing an expression at a particular location, the expression is called the *designated expression.*

Definition An error-based arena $\mathbf{E} = <P,S,D,L,A>$ is *alternate-sufficient* if and only if $\exists$ R ϵ $P_{\mathbf{E}}$ such that R is correct with respect to S.

Alternate-sufficient captures the idea that at least one alternate program is correct. With it testing can sometimes demonstrate correctness. Clearly, there are error-based arenas which are not alternate sufficient. It is more interesting to ask whether this is a decidable property.

Theorem It is undecidable whether an arbitrary error-based arena is alternate sufficient.

Proof Consider the error-based arena $\mathbf{E} = \langle P,S,D,\phi,\phi \rangle$. Deciding whether or not this arena is alterate-sufficient is equivalent to deciding whether P is correct with respect to S. This is, however, undecidable.

The theorem shows that error-based testing must restrict the alternative expressions so that testing can do something "useful". Testing is useful when a finite test set provides enough information to conclude that the specified alternatives could not have been substituted without causing a program failure on the test set. Of course, it is possible to produce an equivalent program by such a substitution. If this occurs, no test data will reveal a difference. This concept of usefulness is embodied in the formal notion of differentiation:

Definition For a program P and $x \in dom([P])$, x *differentiates* P from a program R if and only if

$[P](x) \neq [R](x)$, or
$[R]$ is not defined for x, or
$[P] = [R]$

For a program P and a set $T \subseteq dom([P])$, T *differentiates* P from a program R if

and only if $\exists$ $x \in T$ such that x differentiates P from R. For an arbitrary set of programs **R**, program P, and set T, T *differentiates* P from **R** if and only if for each R in **R** T differentiates P from R.

Differentiation is non-effective because program halting and program equivalence is undecidable. The goal of error-based testing is to determine for each alternate program whether it is equivalent to the original or different. This is achieved by a series of intermediate stages in which the alternate programs are separated into two classes, programs which have been *eliminated* and those which have not. For eliminated programs the test set demonstrates that they cannot be correct by including inputs on which the original and the alternate differ, or the test set has points for which it is proven that the alternate does not halt or is equivalent. Though the proof is not effective, this does not reduce its value. In either case, the eliminated programs are no longer of any interest since they are either incorrect or equivalent to the original. The concept of differentiation provides a convenient means of describing those alternate programs which deserve further attention. These are the programs which agree on the test set yet are not equivalent. Error-based testing is accomplished when all alternate programs are eliminated.

3.2. Differentiating alternate programs

Error-based testing for an error-based arena **E** = <P,S,D,L,A> involves two steps. First, **E** must be alternate-sufficient. For example, it may be known that a certain library program exists and is correct (shown, perhaps, by formal

proof), but its name is forgotten. A test set can be developed which selects the desired program from the library. Another example is when it it known that the specification can be implemented by a program from a given class, as in the case of algebraic program testing [How78]. In general, however, E is merely assumed to be alternate-sufficient. Second, a test set must be constructed that either fails or differentiates P from P_E. This aspect of error-based testing is discussed below.

Definition An error-based arena $E = <P,S,D,L,A>$ is *finite* if and only if $\exists$ a finite test set T for E such that T differentiates P from P_E. If no such test set exists, then E is *infinite.*

Theorem Every bounded error-based arena is finite.

Proof The test set need only include one point to differentiate each non-equivalent alternate program. In a bounded error-based arena, there are a finitely many locations and alternatives, hence there are a finite number of alternate programs. Therefore, the arena is finite.

Error-based testing for a bounded error-based arena is called *mutation testing* [DeM78]. The assumption that the arena is alternate-sufficient is called the *competent programmer hypothesis.* This states that a competent programmer will write a program that is syntacticly close to the correct program, i.e. that the program will fall in a restricted class of programs, one of which is correct. Since the mutation testing arena is bounded, by the above theorem it is finite. Thus, all alternate programs can be generated. The set of alternate programs is

unfortunately huge, and the undecidable program-equivalence problem remains.

One limitation of mutation testing is its restriction to bounded error-based arenas. It is certainly desirable to be able to do error-based testing for unbounded error-based arenas. This, however, cannot be done in general since an unbounded error-based arena can require an infinite test set to differentiate all the alternate programs. Fortunately, this is not always the case.

Theorem There are unbounded error-based arenas which are finite.

Proof As a trivial example, consider

```
proc constant
print 5
endproc
```

Suppose the *alternative expressions* are "any constant" and the location is that of the constant 5. Then the infinite set of alternate programs is the set of all constant programs. Any finite test set (even the empty set) differentiates the original program from all alternate programs.

A second problem with mutation testing relates to the phenomenal number of mutants that can be generated for even a simple program. [DeM78] assumes that simple faults are linked to complex faults in such a way that a test set which differentiates a program with simple faults, differentiates programs containing complex faults. This assumption (dubbed the "coupling effect") reduces the number of alternate programs that must be generated. It is not verifiable since "simple" and "complex" are not precisely defined.

3.3. Coupling

The following definition is a restricted but precise definition of coupling.

Definition For the error-based arena **E** = <P,S,D,L,A>, two expressions at the distinct locations l_i and l_j, ***couple*** on a test set T for **E** if and only if

$(\exists\ f \in A_i)\ (\exists\ g \in A_j)$ such that

(1) P is incorrect with respect to S.
(2) T is successful for P with respect to S.
(3) T differentiates P from $P_f^{l_i}$ and $P_g^{l_j}$
(4) $P_{f,g}^{l_i,l_j}$ is correct with respect to S.

Conditions (1)-(3) state that the program under test is incorrect, yet testing has implied that each of the separate substitutions are incorrect by themselves. Condition (4) says that the substitutions in combination correct the program. This definition can be generalized (at a great loss of clarity) to allow for n expressions.

An example of coupled expressions is given by this program to compute the average of three numbers:

```
proc average
   int i,j,k
   read i, j, k
   ave := 1 + j + k / 3
   print ave
endproc
```

The program illustrates two mistakes by a novice programmer. First is the logical error of the misparenthesization. Second is the typographical error of 1 for i. For input i = 6, j = 2, and k = 7 the program correctly computes 5, and

differentiates both of the alternate programs with single substitutions:

ave := (1 + j + k) / 3

ave := i + j + k / 3

which compute, respectively, 3 and 11. From the data collected, it would be possible to erroneously conclude that the test has eliminated all alternate programs. Thus, the two expressions are coupled.

The general results about coupling are not encouraging.

Theorem It is undecidable whether or not coupled expressions occur in an arbitrary error-based arena.

Proof Take a straight-line program with coupled expressions. Guard the statements by an arbitrary condition. The expressions are still coupled if and only if the path will be executed. This implies the ability to decide whether an arbitrary condition is ever true, which is undecidable.

Corollary It is undecidable whether or not coupled expressions occur in a bounded error-based arena.

However, there is some hope.

Theorem There are unbounded error-based arenas for which the presence of coupled expressions is decidable.

Proof An example is in section 3.4.

Coupling is a pernicious problem that can compromise all error-based testing strategies. In revealing subdomains [Wey80], the problem is cited and then ignored. In domain testing [Whi80] and perturbation testing [Zei83] the problem is presumed not to occur as a "simplifying assumption." As mentioned above, mutation testing identifies the problem, and tries to argue on the basis of a few experiments that coupled expressions rarely occur. In section 3.4 formal arguments are given which indicate that expressions indeed couple infrequently for certain classes of programs. When they do, however, program faults may go undetected.

3.4. The likelihood of coupling

The previous section introduced coupled expressions as the primary pitfall of error-based testing. This section discusses why this is so and describes approaches to analyzing how likely coupling is.

In the simplest case coupling relates four programs determined by error-based testing: P, $P^{l_1}_A$, $P^{l_2}_B$, $P^{l_1,l_2}_{A,B}$. For notational convenience these alternates are denoted by P_A, P_B, and P_{AB}. When locations l_1 and l_2 differ, the possibility of coupling arises.

Definition For alternate program P^l_e and alternative e at $l \in A$ the *associated failure set* is $\{x \mid [P](x) \neq [P^l_e](x)\}$.

Let D_A, D_B, and D_{AB} be the associated failure sets of P_A, P_B, and P_{AB}, respectively. The potential intersections of these sets are shown in figure 1.

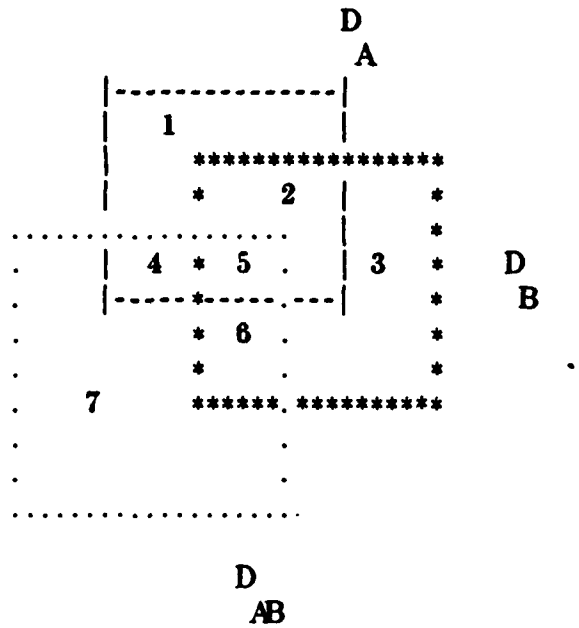


Figure 1

Conditions for coupling can be expressed by assertions about regions of the diagram. For example, it is necessary that

(1) Region 7 be non-empty.

(2) Region 2 be non-empty or regions 1 and 3 must be non-empty.

(3) The test set T be a subset of regions 1-3.

If the existence (or non-existence) of points in one group of regions implies the existence (or non-existence) of points in another region, then perhaps something positive can be said about the absence of coupling. Unfortunately,

Theorem Any combination of regions can be empty.

Proof Let P be a table-lookup program that chooses which table to search depending on whether the variables x and y are assigned 0 or 1. (Thus, table 1 is chosen if x = 0 and y = 0, table 2 is chosen if x = 0 and y = 1, etc.) Let x and y be initialized to 0 in P, and let the alternate programs have the initializations

P_A x = 0, y = 1

P_B x = 1, y = 0

P_{AB} x = 1, y = 1

The effect stated by the theorem can be confirmed by appropriately filling in the tables.

If nothing can be said about the emptiness or non-emptiness of any regions, perhaps something can be said of relative sizes of the regions. In particular for coupling to occur on a test set T it must be that T is a subset of regions 1-3. If therefore it can be shown that these regions are "small" relative to D_{AB}, then the expressions couple on relative "few" test sets. Hence, if the test sets are chosen randomly, there is little chance of selecting a bogus test set.

To investigate the relative sizes of regions, consider an example of coupling in linear equations. Suppose the original program computes

$f(x)=3x+4$

and the correct program computes

$g(x)=4x+3$

Let the two alternate programs compute

$f_1(x)=4x+4$

and

$f_2(x)=3x+3.$

For what values of x will coupling occur? It is given that the first and the fourth conditions of coupling are satisfied (the original is incorrect and the double alternate is correct). To satisfy the second condition, inputs must solve the equation

$f(x)=g(x)$

i.e., the original must compute the correct answer on the test set. The only solution is x = 1. For any other x the equation is not solved. Thus, coupling can only occur on the point x = 1. Coupling does occur for this point because the third condition is satisfied, namely,

$f(1)\neq f_1(1)$

and

$f(1)\neq f_2(1)$

Thus the expressions couple, but only on input 1. Such a point is called a *coupling point.* For this example then the set of coupling points is small compared to the points for which coupling does not occur. There is therefore little chance of randomly choosing 1 out of all possible integers as the point to differentiate the single alternate programs.

In the argument above the exact values of the constants are immaterial. What is relevant is the size of the solution set, not the particular solutions. In fact, the entire argument above can be maintained for the general class of

programs which compute multinomials.

Definition A *multinomial* in x_1 , ... , x_n is any expression in x_1 , ... , x_n and integer constants using only the operators +, -, or *.

Let P be a incorrect program which computes a multinomial in n variables. Suppose the correct program is P′, an alternate program for P produced by substituting several alternatives for expressions in P. Suppose further that P′ computes a multinomial in n variables. Then the solution set to the equation

$$[P](x) = [P'](x)$$

is at most an n-1 dimensional subset of the input space.

Since the class of multinomials includes the classes of linear forms and polynomials, the following theorem is established.

Theorem For any program that computes a linear form, a polynomial, or a multinomial its coupling points are a zero-volume subset of the input space.

The proof of the above theorem is invalid if functions more general than multinomials are allowed. This is because the solution set may not be an n-1 dimensional subset of the input space. For example, for the program that computes

$$f(x)=\frac{2}{x}$$

but should compute

$$g(x)=\frac{4}{3x},$$

every $x > 2$ is in the solution set.

Another possible extension to the above theorem is to allow the introduction of conditional statements to compute selected multinomials. This fails, however, because a condition can arbitrarily restrict a multinomial to only be computed on its coupling points. Of course, the opposite can occur and the path conditions make it impossible for any coupling to occur.

4. Static error-based testing

In this section a form of error-based testing is developed in which properties of the code are proven and later used to judge the quality of a test. Since the technique is code-based it is called *static error-based testing.* A model of computation is first developed to support the discussion of static error-based testing. This model is then applied in a particular method of static error-based testing. The last section presents some undecidable results about this method of testing.

4.1. A Model of Computation

Definition A *state* of a program P is an ordered pair (n,v) where n is a statement number from P and v is a mapping

v: var --> value

which associates a unique value with every variable of P. An *initial state* of P is a state which exists before any of its statements have been executed. A *final state* is a state which exists after P terminates.

The statements of the program are numbered starting at 0. Therefore the initial state is given by (0,v) where v is the mapping which describes how all the variables are initialized. The mapping holds before the associated statement is executed. A psuedo-statement is introduced after each terminal statement to allow the final state to be well-defined.

Some variables (x_1,...,x_n) of the program may be designated as program *input variables*. Their corresponding values (u_1,...,u_n) are called the program *input*. All other variables are initialized to zero. Some variables (y_1,...,y_m) of an initial state may be designated as program *output variables*. Their corresponding values (v_1,...,v_m) upon program termination are called the program *output*. If no variables are explicitly designated as input or output, then all variables are considered both input and output.

The *program function* that a program P computes, denoted by [P], is therefore,

[P] = {(u,v) | v is the output of P on input u,
where u and v are ordered sets of values}

Any arbitrary program segment implicitly defines a program function with all variables designated input and output.

If e is an expression with n variables, and s is a state defined for those variables, then

[e] = { (s,i) | i is the evaluation of expression e
over the variable mapping in the state s}

In an obvious manner differentiate can be extended to include program expressions.

Definition State s *differentiates* expression e from expression f if and only if

[e](s) $\neq$ [f](s) or [e] = [f].

Definition A *path* through a program P is any flow of control through a flowchart of P, assuming that each branch of every condition can be taken. A path which cannot be executed by P is *infeasible*; otherwise, it is *feasible*.

Definition A *computation* of program P beginning with state s = (n,v) is the sequence of states produced by P when started at the statement n with variable values determined by v. This is abbreviated by comp(P,s). If $n \neq 0$ then the computation is *limited*, otherwise it is *full*.

When a program enters an infinite loop, the resulting computation is infinite. A program which terminates abnormally (as in a division by zero) is treated as if it entered an infinite loop. Limited computations may begin in an arbitrary state. This is analogous to loading a program, choosing an arbitrary contents for memory, and starting execution at any statement. Clearly, such computations need not be the tail of a full computation. Two computations are the same if and only if they have exactly the same state sequence.

The output of program P started in state s is written as {P}(s). For a program with one input variable x, [P](x_0) therefore is equivalent to {P}(0,v) when $v(x) = x_0$ and v(y) = 0 for all other variables.

Definition If C is a computation of a program, then prog(C) is the sequence of program statements determined by C. If s is a state in the computation, succ(s) denotes the following state and prev(s) denotes the previous state. Prev and succ are undefined at the beginning and ending of a computation, respectively.

Definition Let $\mathbf{E} = \langle P,S,D,L,A \rangle$ be an error-based arena, and $l_i \in L$. Then the set of *alternate computations* for **E** for location l_i and input x is

$$\{ C \mid C \text{ is the computation of } P_f^{l_i} \text{ for input x, } f \in A_i \}$$

The alternate computations are those computations the program would take if an alternative were substituted for an expression in the program.

4.2. The model applied

By analyzing alternate computations, it is sometimes possible to prove that a test set would differentiate the corresponding alternate programs. A different computation is detectable if and only if a different state is introduced into the computation and the difference persists to the end of the computation. Conditions under which this creation and propagation of different states occurs are defined as follows.

Definition A *creation condition*, B, for alternative set A to expression e at location l in program P is a boolean condition which satisfies

$$\forall S \subseteq S_P, B(S) \Rightarrow (S \text{ differentiates e from A})$$

where S_P denotes all program states of P with location component l.

A creation condition describes program states in which the substitution of non-equivalent alternatives would alter the computation. Consider the case when a creation condition is satisfied by a single state. All non-equivalent alternatives differ from the original expression for this state. Therefore, if any of these

alternatives were substituted, a different computation would result. (This of course assumes that the expression affects the computation by altering the program flow or the variable mapping.) The typical case, though, is for some of the non-equivalent alternatives to differ and others to remain the same for any given state. The definition therefore allows for a creation condition to be satisfied by a set of states, the combination of which differentiates all alternatives. This set of states may result from a single execution in which the source statement is repeatedly executed, or it may result from multiple program executions with different inputs.

By themselves, creation conditions can be used to eliminate alternate programs. For example, a potential fault for a print statement is to reference the wrong variable. A creation condition which guarantees that this fault will not go undetected is "all variables have different values." Using creation conditions for larger code segments is impractical however since the creation condition must be strong enough to ensure that all alternatives are differentiated. Thus, creation conditions are more effective when used to detect the beginning of an incorrect computation. Satisfaction of a creation condition means if a non-equivalent alternative were substituted, a different state would be introduced into the computation. Proving that a particular condition is a creation condition involves showing that a finite test set can differentiate the original expression from all its alternatives in the alternative set.

Creation conditions allow the deduction that the substitution of alternatives will affect the computation. It must then be considered whether the alternate

computations would remain different and yield a test failure. The following definition is given to facilitate the discussion of the original and the alternate computations.

Definition For an error-based arena **E** = <P,S,D,L,A> and for l in L, state(P,l,s) is that member (l,y) of comp(P,s) such that no state (l,x) precedes (l,y) in comp(P,s).

State(P,l,s) therefore is the first state in the computation of P beginning in state s whose location component is l. Imagine the parallel execution of a program and all its alternates induced by an alternative set associated with a program location l. Let all the programs start on the same initial state s. Their computations are all identical until the location l is reached. The potential now exists for some of the computations to diverge. State(P,l,s) denotes the state just prior to the divergence in all these computations. If a creation condition is satisfied at this point, each non-equivalent alternative will produce a different successor state than is produced by the original program. It is then necessary to show that for each pair of states (s_1,s_i) where s_1 is produced by the original program and s_i is produced by the i^{th} alternate program, that propagation occurs:

Definition Let (s_1,s_2) be a pair of states from the computations of a program and alternate for the same input. Let C_1 and C_2 be the ensuing computations. *Propagation* occurs for (s_1,s_2) if and only if

(1) C_1 and C_2 are finite and $\{prog(C_1)\}(s_1) \neq \{prog(C_2)\}(s_2)$, or

(2) Only one of C_1 and C_2 is finite.

Definition Let $E = <P,R,D,L,A>$ be an error-based arena. Let $l \in L$ and let F be the corresponding member in A. A *propagation condition* B for F is a boolean condition which satisfies

$$\forall S \subseteq S_P \times S_P \; B(S) \Rightarrow \forall f \in F \; \exists (s_1,s_2) \in S$$

for which propagation occurs for P and P_f^l

where S_P is the set of all states of P.

Any pair of states that satisfy a propagation condition are guaranteed to produce different outputs. A propagation condition thus ensures that if an alternate state is introduced, the resulting computation guarantees that a failure will occur. The above definition is complicated by the fact that the propagation condition may depend on the particular alternative chosen. If the remaining computation does not encounter that location again, then the definition can be simplified by removing the quantification of alternatives [Mor81].

The following theorem shows that a test set is reliable which satisfies all creation and propagation conditions for a class of alternatives.

Theorem Let $E = <P,S,D,\{l\},\{A\}>$ be an alternate-sufficient error-based arena. Let T be a successful test set for E for which a creation and a propagation condition for A hold. Then P is correct with respect to S.

Proof Since the arena is alternate-sufficient either P or one of its alternates is

correct. Since the creation condition for the alternative set is satisfied for T, all the computations for P on T differ from the computations for the alternate programs which contain non-equivalent alternatives. Satisfaction of the propagation condition implies that propagation occurs for at least one pair of states for each non-equivalent alternate program. Therefore, since the test set is successful all alternate programs are eliminated and the original must be correct.

4.3. Example

To illustrate static error-based testing, several potential faults are eliminated from a program that calculates the area under a curve defined by a polynomial (figure 2).

The arena is formally defined as <P,S,D,L,A> where

P is the program of figure 2,

S = {((a,b,incr),y) | y = area under $4x^2+3x-2$ between the points a and b as estimated by the rectangular method with steps of size h}

D = Dom(S),

L = $\{l_1, l_2, l_3\}$ where

l_1 is the location determined by the expression area in line 11

l_2 is the location determined by the expression incr in line 8

l_3 is the location determined by the expression 4*a*a+3*a-2 in line 2

A = $\{F_1, F_2, F_3\}$ where

F_1 = {a, b, incr, v}

F_2 = {a, b, area, v}

```
program calcarea (input, output);
 var a, b, incr, area, v : real;

 begin

 1    read (a,b,incr);  {incr > 0}
 2    v := 4*a*a + 3*a -2;
 3    area  := 0;

 4    while a + incr <= b do
      begin

 5          area := area + v * incr;
 6          a := a + incr;
 7          v := 4*a*a + 3*a -2;
      end;

 8    incr := b - a;

 9    if incr >= 0 then begin

10      area := area + v * incr;
11      writeln ('area by rectangular method:', area)

      end else

12      writeln ('illegal values for a=', a, ' and b=', b)

 end.
```

Figure 2

$F_3 = P^+(9)$

$P^+(M)$ is the set of polynomials of the form

$a_0 + a_1x^1 + \ldots + a_nx^n$

where $0 \leq a_i \leq M$ and a_i is an integer.

The first two alternative sets represent the fault of substituting one variable for another. The last represents the fault of making a single digit coefficient fault in a polynomial. This set is infinite since there is no restriction on the degree of the polynomial, just the magnitude of the coefficient. To eliminate all alternatives, it is necesary to give a creation condition for each set and to prove a propagation condition to guarantee that any local effect will affect the output. This is done below for each alternative set.

Alternative set F_1. Let S be a set of program states that occur before line 11. A creation condition for alternative set F_1 is

$$\exists\ s \in S\ \forall\ f \in F_1\ [f](s) \neq [area](s)$$

If this condition holds then S differentiates area from F_1 and hence it is a creation condition. Since the statement essentially terminates the program, true is a valid propagation condition; if the wrong value is computed in this statement, it is printed and thus affects the output.

Alternative set F_2. Let S be a set of program states that occur before line 8. A creation condition for this alternative set is

$$\exists\ s \in S\ [incr](s) \neq [b\text{-}a](s)$$

If this condition holds then for all alternatives $f \in F_2$

$$[incr := b\text{-}a](s) \neq [f := b - a](s)$$

since the value of incr will have changed. Thus, the above condition is a creation condition. Let (s_1, s_2) be the pair of states produced by the original program and some alternate program immediately after the creation condition is satisfied. A

propagation condition is

$[b < a](s_1)$ and $[b < a](s_2)$

since this ensures that incr is negative in the original program and positive in each alternate program (incr begins as a positive number and is unchanged until statement 8). Thus every alternate program would execute statement 12 instead of statement 11, yielding a program failure.

Alternative set F_3. Let S be a set of program states that occur before line 2. [Row81] shows that point $x \geq M+1$ is transcendental for the class $P^+(M)$. Thus a creation condition is

$\exists\ s \in S\ [a \geq 10](s)$

since $M = 9$ in this case. A propagation condition for (s_1,s_2) defined as above is

$[a < b < a + incr](s_1)$ and $[a < b < a + incr](s_2)$

which forces the program to follow the path p = (1,2,3,4f,8,9t,10,11) skipping the body of the loop at 4. The condition $a < b$ after line 1 ensures that $incr > 0$ after line 9. Since v remains unchanged along p, the computation of area in line 10 is incorrect since $incr > 0$. Thus, for the above propagation condition, an alternate value in v after line 2 propagates to the output of area in line 11.

Once the creation and propagation conditions are determined, it is necessary to find a test set which causes them to be satisfied. For the error-based arena described above, the following will do:

a = 10, b = 11, and incr = 5

This test point satisfies one creation/propagation pair for each of the specified locations. If the output is correct (no specification has been given) then none of the proposed alternate programs can be correct without the original program being correct. All potential alternate programs described by the alternative sets are therefore eliminated.

4.4. Undecidable problems of static testing

As might be expected, in the presence of multiple alternative locations coupled expressions present many problems.

Theorem Let E = <P,S,D,L,A> be an alterate-sufficient error-based arena with multiple locations. Let T be a successful test set for E for which creation and propagation conditions hold for each alternative set in A. If no set of expressions specified by L couple on T then P is correct with respect to S.

Proof Recall that for a set of expressions to couple four conditions must be satisfied:

(1) P must be incorrect with respect to S.

(2) T must be successful.

(3) T must differentiate P from all single alternate programs.

(4) The multiple alternate program created by the simultaneous substitution of all the single alternative must be correct with respect to S.

For each set of expressions which do not couple for T, one or more of these four conditions must not hold. That the second condition holds is assumed. The third condition holds since the creation and propagation conditions are satisfied on the test set T. If the first condition does not hold, then P is correct and the proof is completed. If the fourth condition does not hold, then the combination of alternatives is an incorrect program. But the arena is alternate-sufficient, so P or one of its alternates is correct. The third condition implies that none of the non-equivalent single alternate programs are correct. Thus, the original program, P, must be correct.

Since creation conditions are dependent on the choice of alternative sets and propagation conditions apply to arbitrary programs the following two theorems come as no surprise.

Theorem There exists no algorithm for generating creation conditions satisfied by finite sets.

Proof For the program

```
input x
print x
```

consider the alternative set defined by replacing "print x" with "print f(x)", where f can be any function computable in the language. Since no finite set can be reliable for this arena, no creation condition can be generated that will be satisfied by a finite set.

Theorem There exists no algorithm for deciding if a non-trivial propagation condition exists.

Proof Consider the case when the propagation condition does not depend on the alternative set, e.g., when the program can be partitioned as

R
location l
P

where there are no branches out of P. Thus, the computation after location l is determined only by P. Since P can be an arbitrary program, if a propagation condition exists for the specified location and corresponding alternative set, then there is a pair of states (s_1,s_2) for which $\{P\}(s_1) \neq \{P\}(s_2)$ But this means P must be defined for some input. This is an undecidable question, however, so the existence of a non-trivial propagation condition is undecidable.

4.5. Conclusions

The following static error-based testing strategy is suggested by the above discussion:

(1) Identify locations within the program where expected faults could occur.

(2) Specify the alternative sets for each of these locations.

(3) For each location:

> a. Derive a condition which asserts that the computation will vary if an alternative is substituted at the given location (a creation condition).

b. Derive a condition which asserts the alternate limited computation(s) will differ from the original computation at their termination (a propagation condition).

c. Satisfy the conditions of (a) and (b) with a test set, then inspect the output. If the output is correct, substitution of non-equivalent alternatives at that location would produce a program failure for the test set.

Some of the advantages of this strategy are:

(1) The strategy extends mutation testing in three ways. First, it allows unbounded error-based arenas; mutation testing does not. This is because a creation condition can differentiate an expression from an infinite set of alternatives. Second, alternate programs are neither generated nor executed since the propagation condition detects when the substitution of an alternative would affect the program output. Thus, the execution overhead of mutation testing is removed. Finally, an executable oracle is not required since the number of test cases are minimized.

(2) The strategy employs formal verification in a useful, limited capacity. To gain the advantages given in (1) it is necessary to develop both creation and propagation conditions. The former are reusable since they are program independent. Development of the latter requires less effort than a verification proof since the full semantics of the program need not be considered. It is only necessary to prove that propagation will occur along chosen paths. If these paths are judiciously selected, the difficulties are minimized.

(3) The strategy provides a means of assessing test data quality. Every test point eliminates potential faults; this strategy allows some of those faults to be determined. This suggests a thorough test is one for which all faults from all alternative sets are eliminated. This parallels Howden's notion of a complete test [How82].

(4) The strategy is partially automatable. Once the creation and propagation conditions are developed, they can be checked by a run-time monitor.

(5) The strategy directs the tester to potential trouble locations. After the test set has been executed, the monitor can give information about alternatives that have not yet been eliminated, which in turn can direct the selection of additional test data to better exercise the designated expression.

The primary disadvantage of the strategy is that creation and propagation conditions cannot be automatically generated.

5. Symbolic Testing

This section presents a dynamic error-based testing strategy called *symbolic testing*. In the first section symbolic testing is formalized and illustrated. Ramifications of the method are then developed for a simple language in the second section. Since error-based testing allows unbounded arenas, undecidable issues are investigated in the next section for the infinite alternative set of constant substitutions. An extension of results for constant substitution to other unbounded arenas is discussed. The last section deals with how to handle problems when multiple paths are involved.

5.1. Motivation and basic definitions

Symbolic execution [Cla77] underlies the idea of symbolic testing. The forte of symbolic execution is its abililty to represent infinitely many executions by a single symbolic execution. Historically this capability has been used to represent the parallel execution of one program on an infinite set of inputs. This is accomplished by using a symbolic input to represent all inputs which follow a given path. In such a system, execution proceeds as usual until an input statement is reached. A symbolic input is then obtained and stored as the value of the appropriate variable. Whenever the variable is referenced, its symbolic value is introduced into the computation. This can result in a symbolic expression being stored in another variable. For example, in

```
read x
y := x + 1
print y
```

if the symbolic input x is read, then the value of y on output is x + 1. A symbolic input represents all inputs that follow a particular path. A symbolic execution system therefore determines the function of any given path by expressing the output in terms of the symbolic input.

Definition A *symbolic arena* **S** = <P,S,D,L,A> is an error-based arena <P′,S,D,L,A> augmented in the following ways:

(1) dom([P]) = dom([P′]) $\bigcup$ an infinite set of *symbolic inputs.*

(2) P = P′, except for the replacement of one expression in P′ with an undeclared variable. Referencing such a variable introduces a unique *symbolic alternative* into the computation.

Definition A *symbolic state* of a program is an ordered pair (n,v) where n is a statement number of P and v is a mapping from variables to values

v: var –> value

where value may be an arbitrary expression involving numeric and symbolic values.

Symbolic testing allows representation of another infinite class of executions: that of an infinite set of alternate programs. This is achieved by creating a program called a *transformation* which represents all alternate programs induced by an alternative set. This transformation is created by replacing a designated expression with a symbolic alternative. The symbolic alternative "stands in" for members of the alternative set. The transformation is then symbolically executed

along the chosen path. For example, in

```
read x
y := x + F
print y
```

the symbolic alternative F may represent all possible constant substitutions. For input 5 the output is 5 + F. Thus, for a given (non-symbolic) input a program containing a symbolic alternative determines the output computed by all represented alternate programs.

Definition Let S = <P,S,D,L,A> be a symbolic arena. For location l in L and variable F not in P, P^l_F is called a *transformation* of P.

A transformation is not an alternate program; rather it represents a class of alternate programs. The transformation P^l_F represents all alternate programs P^l_e, for all e in the alternative set.

If the symbolic form of all alternate computations can be determined for an input x, then this form can be equated to the computation of the actual program, yielding a *propagation equation.* The *solutions* of this equation are those alternatives which when substituted for the designated expression produce alternate programs not differentiated by x, provided the same path is followed.

Definition For x ϵ dom([P]), a *propagation equation* for transformation P^l_F is

$$[P](x) = [P^l_F](x)$$

If x is a symbolic input then the equation is *general* otherwise it is *specific.*

For example, consider the following program to swap two integers:

```
proc name
int x,y

read x read y
x := x + y
y := x - y
x := x - y
print x
print y

endproc
```

which produces on input 2,3:

```
x: 3
y: 2
```

The program reads two integers (2,3) and swaps them without using a temporary.

Suppose that the first assignment statement is wrong; perhaps the reference to y in the assignment statement is incorrect. Replacing this expression with symbolic alternative F yields the program

```
proc name
int x,y

read x,y
x := x + F
y := x - y
x := x - y
print x
print y

endproc
```

Re-executing the program for the same input (2,3) produces:

x: 2 + F - (2 + F - 3)
y: 2 + F - 3

The output encodes the functions computed by all the alternate programs represented by the (unspecified) alternative set. One propagation equation is produced by equating the two values computed for y:

$$2 + F - 3 = 2$$

This simplifies to $F = 3$. Thus, no expression which evaluates to a constant other than 3 could be substituted in the original program and not be differentiated by this test.

Another propagation equation can be generated by equating the two computed values for x:

$$2 + F - (2 + F - 3) = 3$$

This however simplifies to $3 = 3$ and thus no alternate programs can be eliminated based on this test.

Consider some potential alternative sets for the designated expression. If F represents the class of constant substitutions, then no constant other than 3 could be substituted and the output be correct for this input. Thus, all constants other than 3 are eliminated by this test. An additional input is therefore necessary to eliminate this alternate program.

If F represents the class of variable substitutions, then an inspection of the state at the designated expression reveals that no variable other than y has the value 3. Thus the test eliminates this alternative set.

If F represents the class of "variable + constant" faults, then the only possible substitutions are x + 1 and y + 0 since the alternative must evaluate to 3 in the current state, which has x = 2 and y = 3. The second alternative is equivalent to the original expression. An additional input is required to eliminate the first alternate program.

Executing the program for input (8,5) yields the following set of equations:

$$8 + F - 5 = 8$$
$$8 + F - (8 + F - 5) = 5$$

Again the second equation simplifies to 5 = 5. The first equation simplifies to F = 5. This eliminates the class of constant substitutions because no constant can evaluate to 3 on one execution and 5 on another. It also eliminates the class of variable + constant substitutions because the only possible substitution this time is x + (-3) which is not equivalent to the previously determined substitution. Therefore the test set {(2,3), (8,5)} is sufficient to eliminate all the aforementioned alternatives.

The following is an example of a symbolic execution using symbolic inputs. The symbolic inputs correspond in name to the variable to which they are assigned.

```
proc name
int b
int c
int t
read b,c,t
t := b + c*b
b := b*t -c
c := c + b*t
print t
print b
print c
endproc
```

The resulting output is:

```
t: b + (c * b)
b: b * (b + (c * b)) - c
c: c + ((b * (b + (c * b)) - c) * (b + (c * b)))
```

Introducing a symbolic alternative into the first assignment statement

```
t := b + c*F
```

and re-executing yields:

```
t: b + (c * F)
b: b * (b + (c * F)) - c
c: c + ((b * (b + (c * F)) - c) * (b + (c * F)))
```

Note that the output is expressed in terms of the symbolic inputs and the symbolic alternative.

Generating the propagation equation for t yields:

$$b + (c * F) = b + (c * b)$$

thus

$$F = b.$$

Symbolic alternative F and variable b must therefore have the same value.

Thus, only expressions equivalent to b may be substituted. Any alternative which simplifies to b will not be differentiated. Any alternative which does not simplify to b will be differentiated.

The propagation equation for b yields the same conclusion:

$b * (b + (c * F)) - c = b * (b + (c * b)) - c$

thus

$F = b.$

If c were the only output, then evaluation would be harder. After forming the propagation equation and simplifying there are two solutions:

$F = b$

and

$2*b*b*c - c*c + b*c*c*F + b*b*c*c = 0$

For this propagation condition there are two separate solutions. In the latter case F must evaluate so that

$2*b*b*c - c*c + b*c*c*F + b*b*c*c = 0.$

The solution to this is the identity

$F = (-1*b*b*c + c - 2* b*b)/(b*c).$

The division operator in this solution is algebraic and not integer division. Since F must be an integer, the solution holds only for those inputs which produce an even division. Examples of this are when $b = 1$, and $c \in \{-3,1,3\}$. Thus, for these inputs the second solution above would not be differentiated.

5.2. Restricted propagation equations

The difficulty in solving a propagation equation depends on the complexity of operators in the programming language. If an operator is powerful enough to introduce an arbitrary computation then there is no algorithm for solving these equations. An example is the application of an arbitrary function. Suppose the language has a binary APPLY operator which interprets the first parameter as a function and applies it to the second. Since the first parameter is unrestricted, such an operator introduces an arbitrary computation. Therefore a propagation equation involving such operators cannot be solved algorithmically because that could entail deciding when two programs produce the same result. This, of course, is unsolvable. LISP and SNOBOL are languages with such an operator.

For a suitably restricted set of operators, there may be a solution procedure for propagation equations. A language with simple arithmetic operators will now be investigated.

Definition A *rational form* in the variables $x_1, \ldots, x_n$ is an expression in $x_1, \ldots, x_n$ and integer constants using only the operators +, -, *, or / (integer division).

Definition A *polynomial* in x is any expression in x and integer constants using only the operators +, -, or *.

Let **S** = <P,S,D,L,A> be a symbolic arena, and consider the transformation P'_F. Each propagation equation is a rational form r(x,F), where x

is an input and F is a symbolic alternative. If the operation involving x and F are from {+,-,*}, then r(x,F) is a polynomial in F whose coefficients are polynomials in x, and for any fixed x_0, r(x_0,F) is a polynomial in F.

Since propagation equations have simple forms in a language with simple operators, it is useful to know something about the properties of these forms. The most useful is:

Theorem [Iba82]. For any two rational forms r and s over x,

(1) There is an algorithm to determine the cardinality of

$$R = \{x \mid r(x) = s(x)\}.$$

(2) It is decidable whether or not $(\forall x)\ r(x) = s(x)$.

Two consequences of this theorem are noteworthy.

Theorem For every loop-free program P with a single input variable, there exists a program P′ such that P′ contains no conditional statements and for any input x, the sequence of variable mappings produced by P′ is the same as that produced by P, for all variables they share in common.

Proof Here are the translation rules for computing a conditional statement.

Conditional expression	Arithmetic Expression
x = 0	1/(x*x + 1)
x ≠ 0	(2*x*x)/(2*x*x-1) or 1 - (x = 0)
x = k	1/((x-k)*(x-k) + 1)
x > 0	if x = 0 then 0 else if x = -1 then 0 else ((x-1)/x) = 1
x > y	(x-y) > 0

A conditional statement can be simulated by conditionally assigning the desired expression to a variable, as in

```
if x = 5 then                    c := 1/((x-5)*(x-5)+ 1)
    a := z                       a := (1-c)*a + c*z
else
    b := z + x                   b := c*b + (1-c)*(z+ x)
```

If the operators of the language are restricted to {+,-,*,/} then a program without conditional statements computes a rational form. Since any program with only IF statements can be translated into a program without IF's, then all loop-free programs compute rational forms. This establishes the following.

Corollary If P and R are two loop-free programs with a single input variable, which use only the operators {+,-,*,/}, then the equivalence of P and R is decidable.

The restriction to a single input variable is to ensure that the rational form is in one variable (and not, for instance, a multinomial). The difficulty with multiple

input variables is that the program can then contain arbitrary conditional expressions from these input variables. Since the input space is the set of integers, these conditional expressions therefore represent all possible Diophantine equations. There can be no algorithm for solving arbitrary Diophantine equations (it is an unsolvable problem, see below), thus there can be no algorithm for deciding the equivalence of two rational forms over several variables.

Definition For a symbolic arena $\mathbf{S} = \langle P,S,D,L,A \rangle$ and $x \in \text{dom}([P])$, the *path domain* determined by x is

$$\{y \mid \text{prog}(\text{comp}(P,0,x)) = \text{prog}(\text{comp}(P,0,y))\}$$

The path determined by x for program P is

$$path_P(x) = \text{prog}(\text{comp}(P,0,x)).$$

The definition is loose in that it considers two inputs in the same path domain if they execute the same sequence of statements. Thus, two inputs could take different branchs provided identical statements are executed.

How are specific propagation equations related to general propagation equations? Each is developed from one program path. The general propagation equation represents the computation of all inputs in the path domain. Thus, the general equation encompasses the specific equation in that the specific equation results from substituting a member of the path domain for the symbolic input in the general equation.

Theorem The solution set for a general propagation equation is a subset of the

intersection of the solution sets for all the specific propagation equations for the same path.

Proof A solution to a general propagation equation is a solution for any given input. Since a specific propagation equation can be derived from general propagation equation by substituting a literal input for a symbolic input, each solution of the general equation must be a solution of the specific.

Corollary A specific propagation equation for input x may have more solutions than a general propagation equation for the path determined by x.

Corollary There may be more non-differentiated alternatives than indicated by the solution set of a general propagation equation.

5.3. An unbounded arena: constant transformations

An infinite alternative set implies infinitely many alternate programs. It is possible that an arena with such an alternative set is not finite, i.e. that no finite test set exists which differentiates the original program from all its alternates. It would be advantageous to know when this occurs. This problem is discussed below for arenas with restricted programs and alternate sets.

The program class has the following features:

(1) Computation: arithmetic expressions using +,-,*,/ (integer division)

(2) Control: if and while statements

(3) Input/output: The program begins with a read statement for a single variable that is executed once at program invocation. The program terminates with a sequence of print statements that are executed only once when the program halts.

(4) Data: All inputs, outputs, and internal values are integers. Division by zero is treated as a fatal error: no output is produced and the program function is undefined for that point.

The class is general enough to compute all computable functions. Results for this language readily extend to other languages.

The alternative set chosen is that of "all constants." Thus, a program expression may be replaced by an arbitrary constant.

Some of the results below depend on properties of equations of the form

$$m(x_1,...,x_n) = 0$$

where $m(x_1,...,x_n)$ is a multinomial in $x_1,...,x_n$. When solutions must be integers, these are called *Diophantine equations*. Fermat's Last Theorem is a famous example of such an equation: There is no integral solution to

$$a^n + b^n - c^n = 0$$

for $n \geq 3$. To date, no one has proved the theorem. However, it has been shown that there is no algorithm for deciding if an arbitrary Diophantine equation has a solution [Dav73]. The instance of the theorem used here is

Theorem There is no algorithm for deciding if an arbitrary Diophantine equation

in two variables has a solution.

In section 3 it was shown that a bounded arena is finite, i.e. an arena with finite alternative sets requires only a finite test set to differentiate the original program from its alternates. When an arena is unbounded, however, it is possible that no finite test set suffices. An earlier example showed that this is not the case for a very particular (location, alternative set) pair. The discussion below applies to more general circumstances in which unbounded arenas are finite.

Two factors seem to influence whether an arena is decidably finite for constant substitutions. One is the unrestricted use of the division operator. The other is alternatives which affect the program flow. It is shown that the presence of either implies that infinite arenas can result, and that it is undecidable when they do. It is also shown that the absence of both produces finite arenas.

Consider program transformations which can affect the flow of the program.

Definition A transformation, P'_F, of P is *domain independent* for an input x if and only if the conditions along the path taken by x in P evaluate the same as the conditions along the path followed by x in P'_F, for all alternatives represented by F. P'_F is *domain independent* for a set T if and only if it is domain independent for each member of T. If no inputs are specified, then the input set dom([P]) is implied. If a transformation is not domain independent, then it is *domain dependent.*

Domain independence captures the intuitive notion of a program change affecting the flow of control. As might be expected, there is no algorithm for deciding if a transformation is domain independent for any x. There are, nevertheless, algorithms in data flow theory [Hec77] which calculate all variables (and hence all expressions) that have the potential of affecting a given expression. If the given expression is a boolean condition then these algorithms pinpoint those expressions which may affect program flow. Hence they also determine those which cannot affect program flow. The undecidable issue lies in the middle ground--those expressions which have the potential of affecting the program flow, but do not.

For example, consider a symbolic arena with program

```
if x = 1 then y := 1
   else y  := x*x
```

and alternative set "all integers" substituted for the constant 1 in the expression "x = 1". The program transformation for this alternative set is

```
if x = F then y := 1
   else y  := x*x
```

Clearly the transformation is domain dependent. The curious thing is that for every alternate program there is a single input which differentiates it from the original. Furthermore, no input differentiates more than one alternate program. Thus no finite test set exists which differentiates all its alternate programs. The arena is therefore infinite. The following has therefore been proved:

Theorem An arena can be infinite for domain dependent transformations

representing constant substitutions.

Domain dependent transformations can introduce unwanted difficulty for error-based testing. But how great is this difficulty? For instance, even though they can produce infinite arenas, perhaps it is decidable when this occurs. If so, other techniques (such as static error-based testing) can be tried. Unfortunately, this cannot be decided:

Theorem There is no program Q that computes a function satisfying

$$[Q](P,P'_F) = \begin{cases} T & \text{if } \textit{the arena is finite} \\ 0 & \textit{otherwise} \end{cases}$$

where T is a non-empty set which differentiates P from P'_e, for all integers e, P satisfies the program restrictions above, and P'_F represents a class of constant substitutions which are domain dependent.

Proof The existence of Q implies that arbitrary Diophantine equations of two variables are solvable. Since the latter is unsolvable, the former cannot exist.

Let f(x,F) = 0 be an arbitrary Diophantine equation in two variables. Consider the program P

if f(x,F_0) = 0 then print 1 else print 2

where F_0 is a constant. f(x,F_0) is therefore a polynomial in x. Let P'_F be

if f(x,F) = 0 then print 1 else print 2

Applying Q, suppose [Q](P,P'_F) = 0. Then the equation f(x,F) = 0 is solved by infinitely many points. If [Q](P,P'_F) = T; execute P on T. If only 2 is printed then the equation has no solution; otherwise there are solutions and the test

determines all of them.

For domain independent transformations it appears there may be hope of dealing with infinite arenas. Since these transformations do not affect the program control flow, the same path must be followed in the original and all alternate programs. The functions computed by each of these paths is a rational form and the equivalence of such forms was shown to be decidable by a finite test set in section 3. However, the question at hand is whether a finite test set always exists to differentiate a rational form from infinitely many such forms. In section 3 it was shown that a conditional statement can be replaced by assignments using rational forms. Thus, the example above can use rational forms and hence the previous two theorems apply.

Theorem An arena can be infinite for domain independent transformations representing constant substitutions.

Theorem There is no program Q that computes a function satisfying

$$[Q](P,P'_F) = \begin{cases} T & \text{if } \textit{the arena is finite} \\ 0 & \textit{otherwise.} \end{cases}$$

where T is a non-empty set which differentiates P from P'_e, for all integers e, P satisfies the program restrictions above, and P'_F represents a class of constant substitutions which are domain independent.

It appears then that if there is any hope of producing decidable results for unbounded arenas, division must be restricted. The following theorem states that this is exactly the case.

Theorem There is a program Q that computes the function

$$[Q](P,P'_F)=\begin{cases} T & \text{if } \textit{the arena is finite} \\ 0 & \textit{otherwise.} \end{cases}$$

where T is a non-empty set which differentiates P from P'_e, for all integers e, P satisfies the program restrictions above, P'_F represents a class of constant substitutions which are domain independent, and the general propagation equation involves no division operators.

Proof Let f(x,F)=0 be the general propagation equation. For a given input x_0, f(x_0,F) is a polynomial with constant coefficients [Row81]. Let F_0 be the point beyond which all values of F do not satisfy f(x_0,F)=0. (F_0 is 2*M + 1, where M is the maximum magnitude of a coefficient [Row81].) Thus, only values between $-F_0$ and $+F_0$ can satisfy the equation. Therefore the input x_0 eliminates all but a finite number of alternatives. The corresponding specific propagation equations can be generated by substituting into the general propagation equation above. Each of these equations is a polynomial in x. Thus, the solution set for each is finite. The union of these finite solutions is a set S. Any x not a member of S therefore eliminates the remaining alternatives. It is therefore necessary to determine if such an x is a member of the path domain. This can be done since all predicates along the path are rational forms over the input variable and (by a theorem in section 3) solutions sets to rational forms have recursive cardinality. If the cardinality is infinite then such an x is guaranteed to be a member of the path domain since S is finite. If the cardinality is finite, then each member of the path domain can be found by exhaustively trying each input. When the

cardinality of the path domain is reached, it is then decided whether or not any element in the path domain is not in S. If x is in the path domain and not in S then Q should produce T = {x}; otherwise, Q should produce 0.

While the results about constant substitutions are clear, it would be nice to extend them to other infinite sets. The key difference between the set of constants and another infinite alternative set (say expressions of the form ax+b, a and b constants) is that of difficulty of evaluation. The latter require knowledge of the program state for evaluation while the former do not. However there is a link between the two that is important. If a constant is eliminated by a test set, then all expressions which evaluate to that constant are also eliminated. Thus, if a technique (as used in the last theorem above) first eliminates all but a finite set of constants and then eliminates the remaining constants one by one, then the technique can be used to eliminate infinite alternative sets satisfying:

No infinite subset of the alternative set evaluates to the same constant.

(This is because after eliminating all but a finite set of constants, there are at most finitely many alternatives remaining.)

5.4. Problems with domain dependent transformations

The problem with integer division (and its inherent power) is its ability to replace conditionals. One way to restrict this ability without losing division is to replace integer division with another operator that does not decide if the numerator is smaller than the denominator. Let // be defined as

$$a//b = \begin{cases} a/b & \text{if } |a| > |b| \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

This transforms the problem into one of domain dependence since to perform division it is necessary to use a conditional statement:

a / b = if |a| >= |b| then a // b else 0

Symbolic testing can be used to explore the effect of domain dependent transformations in the case of single input variables. The necessary and sufficient conditions that require an infinite test set to differentiate all alternate programs are therefore:

(1) A is infinite.

(2) There is an infinite subset of A, $\{e_1, e_2, ...\}$ with associated failure sets $I_1, I_2, ...$, such that the intersection of all the failure sets is empty.

Note that these conditions do not require an infinite number of alternate paths as evidenced by the earlier example to compute the square of the input (in which the first "1" may be replaced by any constant):

```
read x
if x=1 then y := 1 else y := x*x
print y
```

Symbolic testing can be applied to examples such as these by comparing the functions computed by each path. For this example the two functions are [x*x] and [1]. If an input changes domains then an alternate program fails if the input does not satisfy the equation:

x*x = 1

Thus, the only alternative to "x=1" is "x=-1"; any other alternate program will fail on only one input. Thus, even though the arena is not finite (and hence mutation testing could not be applied), symbolic testing has shown the original program correct.

Sometimes the second condition is trivially false, as in the case

if x=1 then y := 2 else y := x*x

Symbolic analysis yields

x*x = 2

which has no solutions. Thus, if for input 1 the predicate is false in an alternate program, then a program failure is guaranteed. For all alternate programs the predicate is false for the input 1. Hence the set {1} eliminates all alternates.

5.5. Summary

This section has discussed symbolic testing, a form of dynamic error-based testing. The following testing strategy is suggested by the foregoing discussion:

(1) Identify locations within the program where expected faults could occur.

(2) Specify the alternative sets for each of these locations.

(3) Execute the program on a test set and record the paths taken and the output produced for each input. Then, for each of path containing a potential fault location,

a. Replace the expression with a symbolic alternative and perform a symbolic execution down this path.

b. Equate the resulting output with the value computed by the original program for that path. This is the propagation equation.

c. Solve this equation for those values which, if they had been introduced at the fault location would not have produced a different output.

d. From these values deduce what alternatives are yet to be eliminated.

The technique serves both a practical and a theoretical purpose. Its practical purpose is to determine which alternate programs are eliminated by a test set. Solutions to specific propagation equations are alternatives which have not yet been eliminated. Some of the advantages of this strategy are:

(1) The strategy enables full use of the computation of a program to deduce the absence of faults. In the past testing strategies have made limited use of the computation, thus ignoring valuable information from which to deduce the absence of faults.

(2) The strategy exploits the strengths of traditional testing and program verification. Program executions are used to provide the knowledge base from which it can be proven that certain faults are not present.

(3) The strategy is partially automatable. The monitoring of the program and the creation of the propagation equation to be solved is fully automatable. Depending on the equation form, solutions can sometimes be automatically found.

Using general propagation equations, some theoretical aspects of error-based testing have been explicated. In particular, it was shown that in the presence of division operators and and domain dependent transformations it is undecidable whether or not infinite test sets are necessary to differentiate the original program from the alternates. It was also shown that in their absence, finite test sets do exist. This result was extended to other infinite alternative sets. An alternate division operator was suggested to ease the problems of division. Symbolic testing was shown to be useful in analyzing domain dependent transformations.

6. Related work

The theory of error-based testing presented here is strongly influenced by prior work in practical and theoretical testing. This section overviews how these contributions relate to the current framework. The purpose is not to conform all previous work to the present, that would be unjust to their original intent. However, it is informative to see how previous thinking has affected that which described here. It is also hoped that the theory presented here provides insight into previous thought.

The general theory of testing in section 1 is derived from [How76]. the concept of a reliable test set is given there as well as a discussion of its unsolvability. The means of defining specification and correctness is taken from [Mil80]. The approach of using failure sets to discuss essential unsolvabilities in general testing is unique to this thesis.

Many have argued for error-based testing, but none so eloquently as [Wey80]. Error-based testing applies the whole realm of programming knowledge to the task of selecting test data and evaluating results. Thus, information about programming style, error-prone language constructs, programmer idiosyncrasies, and even the hour of the day a program is keypunched is all viable information to guide testing. Evaluation of a test should be based on what errors it shows not to be present in a program.

While the study of error-based testing is motivated by the goal of eliminating errors, the desire here to formally capture and explore this process

has reduced the concept of "error" to "fault." This approach greatly benefited from two testing systems which have used the idea of alternative sets. The first such system, [Ham77], illustrates how a compiler can be used to check the adequacy of a test. The tester supplies both the program and a finite sample of expected input-output behavior. The compiler then ensures that the given behavior is not matched by any "smaller" program. Designated program expressions are substituted with a finite number of alternate expressions which are defined by built-in rules of the compiler. When a program expression is encountered for which alternatives are defined, its alternatives are evaluated and compared with the value of the original expression. Those which evaluate differently are marked. After all test data has been executed, the unmarked alternate expressions are printed by the compiler as warning messages, indicating that either the test input-output pairs are inadequate, or equivalent simpler programs have been found.

This compiler-based system may be viewed as an approximation of static error-based testing. The creation condition is that the finite alternatives must compute a different result. Propagation, however, is assumed and not proven. Therefore an alternate program may be eliminated when it is actually correct because it computes a different internal state than the original.

Mutation testing is the second system based on the concept of eliminating alternatives. In all fairness, adherents of mutation testing do not consider it to be error-based testing. Elimination of faults is not their goal; they view their technique as a program exercising tool which has the agreeable side effect of

ensuring that "complex errors" are not present in a program. Regardless of this orientation, it is clear that mutation testing serves well as a formal model for error-based testing, at least in the limited interpretation of "error" as "fault" taken here.

Mutation testing may be viewed as a static error-based testing strategy in which the program is treated as black box. Thus, in mutation testing creation conditions are applied only at program termination. This has the advantage over compiler-based testing in that there is less chance of rejecting a correct alternate. Its disadvantages are that it requires the full execution of each alternate program (and is therefore quite slow), it does not allow for infinite alternative sets, and it requires executable oracles.

The concept of creation conditions is introduced in [Fos78] and formalized in [How82]. The obvious disadvantage (as evidenced in compiler-based testing) is that alternatives can be rejected which are in fact correct. Propagation conditions are introduced here as a means of rectifying this situation. That propagation is necessary is granted by all. Few, however, show any inclination to prove propagation. The assumptions of domain testing [Whi80] typify this attitude. A predicate is considered adequately tested when inputs are found which results in states which lie "close" to the boundary described by the predicate. This is the creation condition. It is then assumed that coincidental correctness does not occur; i.e., if an input follows a wrong path it will produce a program failure. Therefore, if the creation condition is satisfied, a propagation condition is satisfied automatically since a failure is guaranteed. Only recently

has any emphasis been given to proving propagation, and then only propagation between internal points on a path. In [Zei83] the effect of computational statements on predicates is formally proved. The extension to proving that the effect persists to program output is not taken however.

Though the technique has been available for a decade, the use of symbolic execution as a means of proving propagation is unique here. Limited use of symbolic execution (though apparently unrecognized by the authors) appears in [Zei83] and [Bud80]. The former models the computation of programs whose predicates are linear transformations of the input variables. These linear transformations are nothing more than a restricted form of symbolic execution. The latter proves propagation properties of a restricted class of recursive LISP programs by computing the functions taken by various paths.

As mentioned earlier, coupling is a problem inherent in all error-based strategies. Theoretical work in this area has been limited. In [Bud80] it was shown that restricted decision-table programs and a class of recursive LISP functions are finite arenas and that all alternate programs are eliminated when a specified subclass are eliminated. In [DeM78] an experiment is described in which 22,000 multiple-alternates of a program were generated, in an attempt to find single alternative substitutions which could be differentiated, but not the combination. None were found. An experimental analysis of coupling is fraught with the difficulties of assuring that there is a representative sample space of programs and program faults.

7. Conclusions

The approach taken to the discussion of error-based testing has been theoretically oriented. This orientation has affected the presentation in the following ways.

(1) Error-based testing has been theoretically motivated. The fact that failure sets may not be recursively enumerable leads to the conclusion that testing can only be sufficient when the failure set is restricted.

(2) The concept of "error" has been restricted to that of "fault." In practice, error-based testing may be guided by any characteristic of the program development. It is difficult to formally describe errors in the programming process. Therefore error-based testing has been formalized as fault-based testing.

(3) Many of the theorems are pessimistic. In order to describe the theoretical limitations of error-based testing, the worst-case is often used to show some undecidable result. This does not mean that there can never be any solution–it just means that there is no general solution. The pessimism is clearly seen in the results about coupling and infinite arenas.

Two approaches to error-based testing have been presented, static and dynamic. In fact, both use formal techniques of program verification as well as program executions. They may be viewed as points on a "program-verification" continuum. On one end is formal verification which demonstrates program correctness without executing the program. In static error-based testing the

verification effort is divided into proving creation and propagation conditions and finding inputs which cause them to be satisfied. In some sense, execution of the program has been made part of the proof process. Dynamic error-based testing carries this further by making information derived from the execution the center of interest. Once test information has been collected, it can then be analyzed (e.g., by means of propagation equations). Beyond dynamic testing on this continuum are the various structural program coverage measures [Mil74]. These allow deduction that certain simple faults are not in the program, such as unreachable code. At the far end of the spectrum is conventional testing where input-output pairs are simply compared to a specification. The only information gained here is whether or not certain inputs are in the failure set; no extension can be made to other points in the domain.

Static and dynamic error-based testing are complementary techniques. Though static testing requires an overhead in proving creation and propagation conditions, once proved the assessment of test data is automatable. In dynamic testing the overhead follows the program execution. Therefore, when one technique fails the other has the potential for success. For example, suppose dynamic testing fails to eliminate a certain alternate program. More tests could be executed, but the overhead may be prohibitive. Static testing can then be tried because the cost of additional executions is small relative to the initial overhead of developing the creation and propagation conditions. Likewise static testing may fail, perhaps due to an overly strong creation or propagation condition. Dynamic analysis of some of the test cases may reveal this to be so.

This work can be extended in several ways. Many of the decidability results proved here are dependent upon the operators of a language. Other operators should be investigated, especially in applicative languages. Domain dependent transformations cause difficulties because they can require the analysis of a large number of alternate paths to prove propagation. Techniques such as estimating the likelihood of propagation along a path need to be developed to prune these paths. Finally, before any effective use can be made of the theory presented here, more research must be done on what are likely candidates for alternative sets.

References

[Bra74] Brainerd, Walter S. and Landweber, Lawrence H, *Theory of Computation,* John Wiley & Sons (1974).

[Bud80] Budd, Timothy A., DeMillo, Richard A., Lipton, Richard J., and Sayward, Fredrick G., Theoretical and Empirical Studies on Using Program Mutation to test the Function Correctness of Programs, *POPL,* pp. 220-233 (1980).

[Cla77] Clarke, L. A., A System to Generate Test Data and Symbolically Execute Programs, *IEEE TSE SE-2,* p. 215 222 (Sept. 1977).

[Dav73] Davis, Martin, Hilberts Tenth Problem is Unsolvable, *American Mathematical Monthly 80,* pp. 233-269 (1973).

[DeM78] DeMillo, R. A., Lipton, R. J., and Sawyer, F. G., Hints on Test Data Selection: Help for the Practicing Programmer, *Computer 11,* p. 34 41 (April 1978).

[Fos78] Foster, K., Error Sensitive Test Analysis (ESTA), *Digest for the Workshop on Software Testing and Test Documentation,* (Dec. 1978).

[Gel78] Geller, M., Test Data as an Aid in Proving Program Correctness, *CACM 21,* p. 368 375 (May 1978).

[Ger76] Gerhart, Susan L. and Yelowitz, Lawrence, Observations of Fallibility in Applications of Modern Programming Methodologies, *IEEE TSE SE-2,* 3, (Sept. 1976).

[Goo75] Goodenough, John B. and Gerhart, Susan L., Toward a Theory of Test Data Selection, *IEEE Trans. Soft. Eng. TSE-SE1,* 2, pp. 156-173 (June, 1975).

[Ham77] Hamlet, Richard G., Testing Programs with the aid of a Compiler, *IEEE TSE SE-3,* 4, (July, 1977).

[Hec77] Hecht, Matthew, *Flow Analysis of Computer Programs,* North-Holland Publishing Company (1977).

[Hoa69] Hoare, C.A.R., An Axiomatic Basis for Computer Programming, *CACM 12,* 10, p. 576 585 (Oct. 1969).

[How76] Howden, William E., Reliability of the Path Analysis Testing

Strategy, *IEEE TSE SE-2*, 3, (Sept. 1976).

[How78] Howden, William E., Algebraic Program Testing, *Acta Informatica 10*, (1978).

[How82] Howden, William. E., Weak Mutation Testing and Completeness of Test Sets, *IEEE TSE SE-8*, pp. 371-379 (July 1982).

[IEE83] IEEE,, *IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 729-1983*, IEEE (1983).

[Iba82] Ibarra, S. and Leninger, B., Straight-line Programs with One Input Variable, *SIAM J. of Computing*, (Jan. 1982).

[Jon77] Jones, Neil D. and Muchnick, Steven S, Even Simple Programs Are Hard To Analyze, *JACM*, pp. 338-350 (April 1977).

[Lin79] Linger, R. C., Mills, H. D., and Witt, B. I., *Structured Programming Theory and Practice*, Addison-Wesley (1979).

[Mil74] Miller, E., Paige, M., Benson, J., and Wisehart, W., Structural Techniques of Program Validation, *Digest of Papers COMPCON 74*, pp. 161-164 (Spring 1974).

[Mil80] Mills, Harlan D., Function Semantics for Sequential Programs, *Information Processing 80*, (1980).

[Mor81] Morell, Larry J. and Hamlet, Richard G., Error Propagation and Elimination in Computer Programs, University of Maryland TR-1065, Department of Computer Science (July, 1981).

[Row81] Rowland, John H. and Davis, Philip J., On the Use of Transcendentals for Program Testing, *JACM 28*, 1, pp. 181-190 (Jan. 1981).

[Tsi70] Tsichritzis, D., The Equivalence Problem of Simple Programs, *JACM 17*, 4, pp. 729-738 (Oct. 1970).

[Wey80] Weyuker, Elaine J. and Ostrand, Thomas J., Theories of Program Testing and the Application of Revealing Subdomains, *IEEE TSE SE-6*, 3, pp. 236-246 (May 1980).

[Whi80] White, Lee J. and Cohen, Edward I, A Domain Strategy for Computer Program Testing, *IEEE TSE SE-6*, 3, pp. 247-257 (May 1980).

[Zei83] Zeil, Steven J, Testing for Perturbation of Program Statements, *IEEE TSE SE-9*, p. 335 346 (May 1983).

END

FILMED

8-84

DTIC